University Library



A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Amadini, R;Gange, G;Schachte, P;Sondergaard, H;Stuckey, P

Title:
Abstract interpretation, symbolic execution and constraints

Date:
2020

Citation:
Amadini, R., Gange, G., Schachte, P., Sondergaard, H. & Stuckey, P. (2020). Abstract interpretation, symbolic execution and constraints. de Boer, F (Ed.). Mauro, J (Ed.). Recent Developments in the Design and Implementation of Programming Languages, (1), pp.7:1-7:19. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Persistent Link:
https://hdl.handle.net/11343/277037

Abstract Interpretation, Symbolic Execution and Constraints

Roberto Amadini
University of Bologna, Italy
https://www.unibo.it/sitoweb/roberto.amadini/en
roberto.amadini@unibo.it

Graeme Gange
Monash University, Clayton, Australia
https://research.monash.edu/en/persons/graeme-gange
graeme.gange@monash.edu

Peter Schachte
The University of Melbourne, Australia
https://people.eng.unimelb.edu.au/schachte/
schachte@unimelb.edu.au

Harald Søndergaard
The University of Melbourne, Australia
https://people.eng.unimelb.edu.au/harald/
harald@unimelb.edu.au

Peter J. Stuckey
Monash University, Clayton, Australia
https://research.monash.edu/en/persons/peter-stuckey
peter.stuckey@monash.edu

Abstract

Abstract interpretation is a static analysis framework for sound over-approximation of all possible runtime states of a program. Symbolic execution is a framework for reachability analysis which tries to explore all possible execution paths of a program. A shared feature between abstract interpretation and symbolic execution is that each – implicitly or explicitly – maintains constraints during execution, in the form of invariants or path conditions. We investigate the relations between the worlds of abstract interpretation, symbolic execution and constraint solving, to expose potential synergies.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Invariants; Software and its engineering → Software maintenance tools; Software and its engineering → Software testing and debugging

Keywords and phrases Abstract interpretation, symbolic execution, constraint solving, dynamic analysis, static analysis

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.7

Acknowledgements We wish to thank the anonymous reviewers for their detailed and constructive suggestions.

1 Introduction

The *abstract interpretation* framework proposed by Cousot and Cousot in the 1970s [14, 15] provides an elegant and generic approach for static analysis. Under certain reasonable assumptions, this method is guaranteed to terminate with a sound abstraction of *all* the possible program traces.


© Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey;
licensed under Creative Commons License CC-BY



Since different information is pertinent in different contexts and applications, each analysis specifies the abstraction of the computation state to use: the *abstract domain* of the analysis. Each element of the abstract domain is an abstract value that approximates a set of "concrete" values, i.e., values that a variable can take during the program execution.

The most common form of abstract interpretation mimics forward program execution in such a manner that it eventually *over-approximates* the set of possible run-time states, for all possible input values, with corresponding abstract values. Abstract interpretation thus provides a method for *invariant generation* – it produces, in finite time, a valid invariant for each program point, including the start and the end of loop bodies. Whether these invariants are useful for whatever task is at hand may depend on the granularity of the chosen abstract domain.

The first *symbolic execution* tools [8, 25] were developed around the same time as the abstract interpretation framework. The main idea behind symbolic execution is to use symbolic expressions instead of concrete values to explore the possible program paths and reasoning about the conditions under which the program execution will branch this way or that. The constraints leading to a particular path being taken are called *path conditions*, so that a given path is feasible if and only if the corresponding path condition is satisfiable. This enables symbolic execution to perform *reachability* analysis.

The early work on symbolic execution saw the technique as an important aid in the systematic testing and debugging [8, 25] and those applications remain the most common. However, it has been noted that Burstall's technique [9] for proving total correctness of programs also involves the use of symbolic execution. (Burstall used the term "hand simulation", and his technique has since been referred to as the "sometime" method, and also as the "intermittent assertion" method.) The method associates, with a program point p, assertions of the form "control will, sometime during execution, reach p with the program state satisfying φ" (note that there is no claim that this will be the case *every* time control reaches p). Showing that a program terminates and satisfies a specification φ thus boils down to being able to associate φ (or something that entails it) with each exit point of the program. As with the more commonly known invariant assertion method, the intermittent assertion methods relies on the discovery of suitable lemmas and their proof by induction. Unlike the invariant assertion method, it can establish *total* correctness.

In this paper we mainly have the less ambitious "debugging" use of symbolic execution in mind. We note, however, that there is continued development of deductive verification systems that utilise the idea behind Burstall's method and its application of symbolic execution. We discuss such systems in Section 6. One, the KeY project [1], is of particular interest, as it extends the symbolic execution mechanism with an invariant generation ability, using ideas from abstract interpretation.

At first glance, symbolic execution may seem very similar to abstract interpretation. Both are methods for abstracting the runtime behaviour of a program across all its possible input values. However, while abstract interpretation is naturally considered a *static analysis*, we contend that symbolic execution is for all intents a *dynamic analysis* for a number of reasons. First, symbolic execution always executes the target program, even if symbolically, in a forward manner. Second, symbolic execution cannot by itself guarantee that *all* the possible paths are covered. Unless aided by some external oracle, symbolic execution *under-approximates* the set of possible runtime states with a number of path conditions. As a dynamic analysis, symbolic execution can produce *witnesses* of fault, but it offers weak guarantees for coverage and termination. In particular symbolic execution runs the risk of "getting caught" in loops. Verification tools based on symbolic execution make use of a variety of techniques to remedy the situation, such as requiring users to suggest invariants.

In contrast, static analyses produce *alarms*, including false alarms, they tend to rely on *intrinsic* conditions and, in principle at least, provide strong guarantees for termination and coverage. Because dynamic analysis focuses on (possibly long) program paths, it is able to find relations among program entities that may be textually far apart. Static analysis instead usually performs detailed, but *local*, analysis, detailed only within basic blocks or functions.

A connecting point between abstract interpretation and symbolic execution is that they both – implicitly or explicitly – collect and solve a number of *constraints* along their execution.

For abstract interpretation, the constraints are implicitly collected during the abstract execution in the form of invariants, which are relations over the program variables. The constraint perspective becomes more evident when we have *relational* abstract domains involving different variables. For example, if we use the polyhedra [16] or the octagon [27] abstract domain we explicitly collect and update linear constraints over the program variables.

The relation between symbolic execution and constraint solving is more straightforward. The path conditions collected by the symbolic engine are constraints over the variables occurring at each branching point of the program. The test for satisfiability is delegated to a *constraint solver*. Its role is crucial for symbolic execution, because its efficiency and expressiveness can strongly affect the performance of the program analysis. As we shall see in Section 3, this is even more true for *concolic testing*, a hybrid technique based on symbolic execution where the constraint solver is used to generate the next input to test according to the last path condition explored.

In this paper we study the relationship between abstract interpretation, symbolic execution and constraint solving. Based on a small Turing complete language, $\mathcal{L}$, we first define the semantics of symbolic execution over $\mathcal{L}$ by specifying how constraints are collected, updated and solved during the execution. Then, after describing the abstract interpretation of $\mathcal{L}$, we show how these techniques are complementary and can help each other to get an overall better program analysis. In particular, we focus on how abstract interpretation may help symbolic execution escape loops through simple program transformations.

Paper structure: Section 2 gives some technical background notions. In Section 3 we explain symbolic (and concolic) execution, and in Section 4 we cover abstract interpretation. Section 5 discusses the relationships between the above techniques, before reporting the related literature in Section 6 and concluding in Section 7.

2 Preliminaries

2.1 Constraint solving

As with abstract interpretation and symbolic execution, the theory of constraint solving dates back to the 1970s [28, 26]. Although there is not a univocal definition, we can informally refer to constraint solving – or constraint satisfaction – as the process of finding a solution to a problem whose variables are subject to a number of constraints restricting their domains. More formally, we can define constraint solving as the process of finding a solution to a *constraint satisfaction problem* (CSP), which is a triple $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of domains, where domain D_i contains the possible values that x_i can take for $i = 1, \ldots, n$; $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints over the variables of $\mathcal{X}$.

Finding a solution of $\mathcal{P}$ means finding a consistent assignment of domain values to variables. Formally, a solution is a map $\xi : \mathcal{X} \to \bigcup \mathcal{D}$ such that $\xi(x_i) \in D_i$ for $i = 1, \ldots, n$ and $C(\xi(x_{i_1}), \ldots, \xi(x_{i_k}))$ holds for each constraint $C \in \mathcal{C}$ over variables $x_{i_1}, \ldots, x_{i_k}$. If $\mathcal{P}$ admits a solution ξ, then it is called *satisfiable* and we will write $\xi \models \mathcal{P}$. If no solution exists,

then $\mathcal{P}$ is *unsatisfiable* and we will write $\mathcal{P} \models \bot$. Note that $\mathcal{P} \not\models \bot$ denotes a satisfiable problem $\mathcal{P}$ without specifying a solution for it (i.e., $\mathcal{P} \not\models \bot$ iff there exists $\xi : \mathcal{X} \to \bigcup \mathcal{D}$ such that $\xi \models \mathcal{P}$). As a small abuse of notation, we extend this notation to constraints: if $C \in \mathcal{C}$ is defined over variables $x_{i_1}, \ldots, x_{i_k}$, then $\xi \models C$ means that $C(\xi(x_{i_1}), \ldots, \xi(x_{i_k}))$ holds, while $C \not\models \bot$ (resp., $C \models \bot$) means that C is satisfiable (resp., unsatisfiable).

The definition of CSP is very general, because no limits are posed on the type of the domains (e.g., they can be integers, reals, rationals, strings, arrays, and so on), or on the type of the constraints to be solved. According to the type of domain and constraints, and the way constraints are solved, different paradigms have been proposed, e.g., Boolean satisfiability/satisfiability modulo theory [7], constraint programming [31], linear programming [17], and so on.

2.2 Abstract interpretation

Abstract interpretation is a framework for the sound over-approximation of program computations. Let $\mathbb{S}$ be the set of concrete values that a program variable can take (e.g., integers, floating point, strings, ...) in any possible concrete execution. The *concrete domain* $\mathcal{C} = \mathcal{P}(\mathbb{S})$ is defined as the powerset of $\mathbb{S}$, and a sound abstraction of $\mathcal{C}$ is given by an *abstract domain* $\mathcal{A}$ for $\mathcal{C}$ and a concretization function $\gamma : \mathcal{A} \to \mathcal{C}$ inducing a partial order $\sqsubseteq$ over $\mathcal{A}$ such that $a \sqsubseteq a' \iff \gamma(a) \subseteq \gamma(a')$ for each $a, a' \in \mathcal{A}$. Typically, a domain $\mathfrak{A}$ is equipped with an order that makes it a *lattice* $\mathfrak{A} = \langle \mathcal{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$, where $\sqcup$ and $\sqcap$ are the meet and join operations, respectively, according to $\sqsubseteq$, and $\bot$ and $\top$ are unique least and greatest elements of $\mathcal{A}$, respectively[1]. Choosing an abstract domain is a compromise between its *precision* (how faithfully it can approximate a concrete domain) and the computational cost of conducting the analysis with it.

The "abstract lattice" $\mathfrak{A}$ is typically connected to the "concrete lattice" $\mathfrak{C} = \langle \mathcal{C}, \subseteq, \cap, \cup, \emptyset, \mathbb{S} \rangle$ via an abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$ mapping concrete elements to corresponding abstract elements. The pair (α, γ) often forms a *Galois connection*, i.e., $\alpha(C) \sqsubseteq a \iff C \subseteq \gamma(a)$ for each $a \in \mathcal{A}$, $C \in \mathcal{C}$. Having a Galois connection corresponds to the existence of a unique *best abstraction* for each $C \in \mathcal{C}$.

Termination of abstract execution can be guaranteed, even in the presence of loops. In some cases, so-called *widening* operators [12] are required to achieve this (or sometimes just to accelerate convergence). A widening operator ∇ for abstract domain $\mathcal{A}$ satisfies two conditions: *(i)* $a, a' \sqsubseteq a \nabla a'$ for any $a, a' \in \mathcal{A}$, and *(ii)* for any sequence $a_0, a_1, a_2, \cdots \in \mathcal{A}$ the sequence $b_0, b_1, b_2, \ldots$ with $b_0 = a_0$ and $b_i = b_{i-1} \nabla a_i$ for $i > 0$ is ultimately stationary, i.e., there does exist $k \in \mathbb{N}$ such that $b_i = b_k$ for each $i \geq k$. In practice, widening allows us to "short-cut" infinite ascending chains, guaranteeing that a (post-) *fixpoint* is eventually reached.

Abstract domains can also be *combined*. Given $n > 1$ abstract domains $\langle \mathcal{A}_i, \sqsubseteq_i, \sqcap_i, \sqcup_i, \bot_i, \top_i \rangle$ abstracting a concrete domain $\mathcal{C}$ with abstraction functions $\alpha_i : \mathcal{C} \to \mathcal{A}_i$ and concretization functions $\gamma_i : \mathcal{A}_i \to \mathcal{C}$ for $i = 1, \ldots, n$, their *direct product* is the structure $\langle \mathcal{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$ where:

- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$
- $(a_1, \ldots, a_n) \sqsubseteq (a'_1, \ldots, a'_n) \iff a_1 \sqsubseteq_1 a'_1 \wedge \cdots \wedge a_n \sqsubseteq_n a'_n$
- $(a_1, \ldots, a_n) \sqcap (a'_1, \ldots, a'_n) = (a_1 \sqcap_1 a'_1, \ldots, a_n \sqcap_n a'_n)$
- $(a_1, \ldots, a_n) \sqcup (a'_1, \ldots, a'_n) = (a_1 \sqcup_1 a'_1, \ldots, a_n \sqcup_n a'_n)$
- $\bot = (\bot_1, \ldots, \bot_n)$ and $\top = (\top_1, \ldots, \top_n)$

[1] One can also find examples of non-lattice abstract domains [19].

A drawback of the direct product is that γ may not be injective, even when all of the γ_i are injective. Its use may give rise to sub-optimal precision, although it does not threaten soundness of the analysis. For optimal precision, the *reduced product* $\mathcal{A}' = \mathcal{A}_1 \otimes \cdots \otimes \mathcal{A}_n$ is required [15]. Informally, $\mathcal{A}'$ removes redundant tuples from $\mathcal{A}$. More formally, $\mathcal{A}'$ is the quotient set of the equivalence relation $\equiv$ on $\mathcal{A}$ such that $(a_1, \ldots, a_n) \equiv (a'_1, \ldots, a'_n) \Leftrightarrow \gamma(a_1, \ldots, a_n) = \gamma(a'_1, \ldots, a'_n)$. This ensures that the resulting γ is injective. For example, if $\mathcal{A}_1$ is the parity domain and $\mathcal{A}_2$ is the interval domain, then $(Odd, [0, 2])$ and $(Odd, [1, 1])$ are different elements of $\mathcal{A}_1 \times \mathcal{A}_2$, yet they have the same meaning. In $\mathcal{A}_1 \otimes \mathcal{A}_2$ they are considered one and the same.

While the reduced product has a simple mathematical definition, it can be difficult or cumbersome to implement. A common alternative is to use *ad hoc* channelling functions to refine pairs of abstract elements, with no guarantee of optimal reduction. Care is needed when widening is defined on reduced products: The distributed operation $(a_1 \nabla_1 a'_1, \ldots, a_n \nabla_n a'_n)$ combining the widening operators of the individual domains is not necessarily a valid widening operation.

2.3 Language $\mathcal{L}$

We now define a simple language, $\mathcal{L}$, that we shall use in Sections 3 and 4 to illustrate how symbolic execution and abstract interpretation work.

We denote by Var the set of all the variables that can occur in an $\mathcal{L}$ program, and with Val the set of values that a variable of Var can take (e.g., Val may contain integers, floating point numbers, string literals, and so on). We assume that values 0 and 1 (which may represent falsehood and truth) always belong to Val. We denote by Loc the set of all the locations, or program points, for a program written in $\mathcal{L}$.

A *concrete state*, or runtime state, is a map $Var \to Val$ from variables to values. A *concrete trace* $\tau : Var \times Loc \to Val \cup \{\bot, \top\}$ is a function returning the concrete state $\tau(x, \ell)$ of variable x at program point ℓ in a given execution of the program. If $\tau(x, \ell) = \bot$, then either ℓ is unreachable or x is not defined at ℓ. If $\tau(x, \ell) = \top$, then the value of x at ℓ is unknown.

Let Fun be the set of all the possible functions of $\mathcal{L}$, i.e., the allowed operations over the variables and values of $\mathcal{L}$. We denote by $Fun^k \subseteq Fun$ the set of functions having arity k, and with $BFun$ the set of the Boolean-valued functions (predicates), that is, the functions that yield values in $\{0, 1\}$ only. The set of expressions Exp over $\mathcal{L}$ is recursively defined by:

$$Val, Var \subseteq Exp$$
$$f \in Fun^k,\ e_1, \ldots, e_k \in Exp \implies f(e_1, \ldots, e_k) \in Exp$$

We define the set $Bool \subseteq Expr$ of the Boolean expressions as follows:

$$0, 1 \in Bool$$
$$b(e_1, \ldots, e_k) \in Exp,\ b \in BFun \implies b(e_1, \ldots, e_k) \in Bool$$

A Boolean expression $b \in Bool$ defines a constraint: if b can evaluate to 1, the corresponding constraint is satisfiable; otherwise, its *negation* $\neg b = 1 - b$ is satisfiable (this however does not imply that b is unsatisfiable).

Now we can define the BNF syntax of the statements of $\mathcal{L}$ as:

$$S ::= \textbf{skip} \mid x \leftarrow e \mid x \leftarrow \top \mid S_1; S_2 \mid \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \mid \textbf{while } b \textbf{ do } S \textbf{ od}$$

where $x \in Var$, $e \in Exp$, $b \in Bool$, and S, S_1, S_2 are statements.

We are deliberately vague about the values in *Val* and the functions in *Fun* because we aim to provide a high-level view of the semantics of the symbolic execution and abstract interpretation of $\mathcal{L}$ without going into too much detail.

3 Symbolic Execution

Symbolic execution was introduced with the aim of describing in a compact way the inputs causing each part of a program to be executed or "covered". In this section we shall see how symbolic (and concolic) execution works for $\mathcal{L}$ from Section 2.

A peculiarity of $\mathcal{L}$ is that it allows the definition of assignments of the form $x \leftarrow \top$. We use this to indicate that the value of x is unknown. It corresponds to an annotation required for symbolic execution, to mark a program variable x as "symbolic". After $x \leftarrow \top$ a Boolean expression containing x will be evaluated in terms of *all* the possible values that x can take in a *particular* concrete trace. In the interest of generality we make no assumption about the domain of a symbolic variable.

Rather than maintaining concrete traces, the symbolic execution of an $\mathcal{L}$ program defines *symbolic traces* keeping track of: 1) the *path* π describing the evolution of the program execution; 2) a *symbolic state* σ mapping variables to (symbolic) expressions; 3) a constraint ϕ denoting a *path condition* for π, i.e., a necessary and sufficient condition, on symbolic values, for execution to follow path π. Formally, a symbolic trace is a triple (π, σ, ϕ) where:

1. $\pi \in (\mathit{Loc} \times \{0,1\})^*$ is a tuple of branch points of the form $\langle \ell_1^{b_1}, \ldots, \ell_k^{b_k} \rangle$ where ℓ_i is the location of the i-th branch point encountered along the execution path, and b_i is either 0 or 1 depending on whether the corresponding condition evaluated to false or true respectively.
2. $\sigma : \mathit{Var} \to \mathit{Exp}$ maps variables to expressions. We extend σ to expressions in the natural way by defining $\overline{\sigma} : \mathit{Exp} \to \mathit{Exp}$ as:

$$\begin{aligned} \overline{\sigma}(c) &= c && \text{for each } c \in \mathit{Val} \\ \overline{\sigma}(x) &= \sigma(x) && \text{for each } x \in \mathit{Var} \\ \overline{\sigma}(f(x_1, \ldots, x_k)) &= f(\overline{\sigma}(x_1), \ldots, \overline{\sigma}(x_k)) && \text{for each } k \in \mathbb{N}, f \in \mathit{Fun}^k \end{aligned}$$

3. $\phi = C_1 \wedge \cdots \wedge C_k$ is a conjunction of constraints denoting the path condition of π; each C_i is either a Boolean expression (if $b_i = 1$) or a negated Boolean expression (if $b_i = 0$) describing the direction taken at each branch point.

A natural way to capture the semantics of a symbolic execution is with a *structural operational semantics* (SOS) representing how the symbolic trace evolves. Fig. 1 shows the SOS definition of $\mathcal{L}$'s semantics. The initial symbolic state is always $(\langle\rangle, \emptyset, \top)$, where $\langle\rangle$ is the empty path and $\top$ indicates a constraint that is always true. We identify σ with the set of pairs $\{x \mapsto e \mid \sigma(x) = e\}$.

The rules in line 1 of Fig. 1 are the usual rules for the skip statement and statement sequencing.

The rules in line 2 handle variable assignment. The assignment $x \leftarrow e$ where $e \in \mathit{Expr}$ simply replaces the entry for x in σ with the expression resulting from the evaluation of $\overline{\sigma}(e)$. The assignment $x \leftarrow \top$ enables variable x to be treated as symbolic: in this case a fresh symbolic value $\widetilde{x}$ is assigned to x. As we shall see, the constraints of ϕ are actually constraints over these symbolic values. Because no branching point is encountered, π and ϕ remain unchanged.

$$\frac{}{\langle \mathbf{skip}, (\pi, \sigma, \phi)\rangle \to (\pi, \sigma, \phi)} \qquad \frac{\langle S_1, (\pi, \sigma, \phi)\rangle \to (\pi', \sigma', \phi')}{\langle S_1; S_2, (\pi, \sigma, \phi)\rangle \to \langle S_2, (\pi', \sigma', \phi')\rangle} \tag{1}$$

$$\frac{\sigma'(v) = \begin{cases} \sigma(x) & \text{if } v \neq x \\ \overline{\sigma}(e) & \text{if } v = x \end{cases}}{\langle x \leftarrow e, (\pi, \sigma, \phi)\rangle \to (\pi, \sigma', \phi)} \qquad \frac{\sigma'(v) = \begin{cases} \sigma(x) & \text{if } v \neq x \\ \widetilde{x} & \text{if } v = x \end{cases}}{\langle x \leftarrow \top, (\pi, \sigma, \phi)\rangle \to (\pi, \sigma', \phi)} \tag{2}$$

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell)\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, (\pi, \sigma, \phi)\rangle \to \langle S_1, (\pi', \sigma, \phi')\rangle} \tag{3}$$

$$\frac{\phi \wedge \neg\overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \neg\overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^0}{\langle (\ell)\ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}, (\pi, \sigma, \phi)\rangle \to \langle S_2, (\pi', \sigma, \phi')\rangle} \tag{4}$$

$$\frac{\phi \wedge \neg\overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \neg\overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^0}{\langle (\ell)\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}, (\pi, \sigma, \phi)\rangle \to (\pi', \sigma, \phi')} \tag{5}$$

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell)\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}, (\pi, \sigma, \phi)\rangle \to \langle S; (\ell)\ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{od}, (\pi', \sigma, \phi')\rangle} \tag{6}$$

Figure 1 Semantics of symbolic execution.

The rules in lines 3–4 show the semantics of the "if-then-else" statement. In line 3, we first check if the constraint $\phi \wedge \overline{\sigma}(b)$ is satisfiable with a suitable *constraint solver*. If so, the "then" branch is feasible and the constraint $\overline{\sigma}(b)$ is added to ϕ. In this case, given that ℓ is the location of the "if-then-else" statement, we also update π with the path π' obtained by appending ℓ^1 to π (we use $\oplus$ to denote the append operation). The map σ remains unchanged, because no symbolic variable is updated.

The rule in line 4 for the "else" branch is totally symmetric: we just consider $\neg\overline{\sigma}(b)$ instead of $\overline{\sigma}(b)$ and append ℓ^0 instead of ℓ^1. Note that symbolic execution is *nondeterministic* in the sense that at each branch point we can follow both branches. Indeed, in general, given constraint C we might find both an assignment ξ satisfying C and one ξ' satisfying $\neg C$.

The rules in lines 5–6 give the semantics of "while" loops. Rule 5 is basically the same as rule 4, while rule 6 is similar to rule 3: the difference is that here we can execute the loop an arbitrary number of times. This is one reason to consider symbolic execution *dynamic* (unless it is somehow enriched with an oracle or some mechanism for inductive reasoning): unlike static analysis, it may get stuck in loops. The applications to debugging and test generation therefore make use of termination criteria based on resource consumption.

▶ **Example 1.** Fig. 2 shows a simple $\mathcal{L}$ program, where $\langle StmtA\rangle$ and $\langle StmtB\rangle$ are unspecified statements. Variable y is symbolic, so before the while loop the symbolic trace is $(\langle\,\rangle, \{x \leftarrow 0, y \leftarrow \widetilde{y}\}, \emptyset)$. The $y > 0$ condition of the while loop at location ℓ_1 (line 3) is then evaluated.

On the one hand, we invoke a constraint solver to check if $\neg\overline{\sigma}(y > 0) = \neg(\sigma(y) > \sigma(0)) = \neg(\widetilde{y} > 0) = \widetilde{y} \leq 0$ is satisfiable (see rule 5 of Fig. 1). Assuming that $\widetilde{y}$ has a numeric domain with a lower bound smaller than or equal to 0, this constraint is clearly satisfiable (e.g., $\widetilde{y} = 0$ is a solution) so ℓ_1^0 is added to π, $\widetilde{y} \leq 0$ is added to ϕ and we skip the body of the loop.

```
x ← 0;
y ← ⊤;
while y > 0 do                    ▷ Location ℓ1
    x ← x + 1;
    y ← y − 2;
od;
if x > 1117 then                  ▷ Location ℓ2
    ⟨StmtA⟩;
else
    ⟨StmtB⟩;
fi
```

Figure 2 A simple $\mathcal{L}$ program.

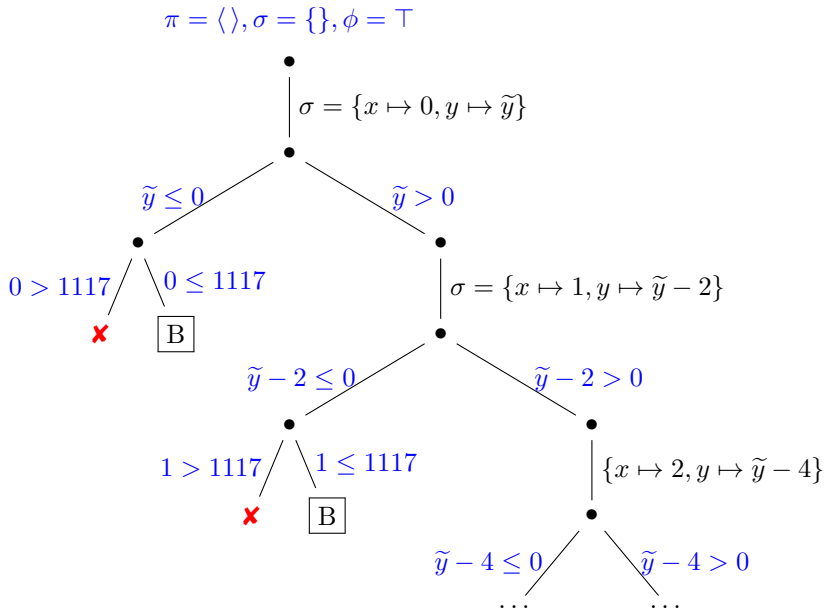


Figure 3 Top part of the symbolic execution tree for the program in Fig. 2.

On the other hand, we also consider the case where the $y > 0$ condition holds: we invoke the solver to check if $\overline{\sigma}(y > 0) = \widetilde{y} > 0$ is satisfiable and we keep on iterating the loop (see rule 6 of Fig. 1). Clearly, if the domain of $\widetilde{y}$ has no upper bound and no halting criterion is set (e.g., a timeout or a maximum number of loops iterations) the computation will not converge. This can be seen from the rightmost branch of the symbolic execution tree in Fig. 3, where the constraint $\widetilde{y} - 2k > 0$ is repeatedly added to ϕ after the k-th evaluation of the while loop condition, corresponding to a k-length path $\langle \ell_1^1, \ell_1^1, \ldots, \ell_1^1 \rangle$.

The if-then-else statement at location ℓ_2 is then processed using rules 3 and 4 of Fig. 1. In the first case, we try to solve the constraint $\phi \wedge \overline{\sigma}(x > 1117)$, where ϕ is the path condition at location ℓ_2. This constraint can only be satisfied if the while loop is executed $k > 1117$ times and $\widetilde{y} - 2k \leq 0$. This invariant cannot be inferred by symbolic execution, hence to find that $\langle StmtA \rangle$ is reachable the symbolic interpreter has to go through the while loop at least 1118 times, hoping that the resource limit is not reached earlier.

Similarly, we apply rule 4 and solve $\phi \wedge \neg\overline{\sigma}(x > 1117)$ to reach $\langle StmtB \rangle$. ⌟

As mentioned, the constraints of ϕ form a path condition because they refer to the conditions under which a path π is feasible, i.e., π is feasible if and only if ϕ is satisfiable. It is important to note that symbolic execution welcomes the definition of branch points and program paths, but in general has no notion of program points. If we define $loc(\pi)$ as the set of all the program points traversed along the execution of path π, a path condition ϕ for π is a *sufficient* condition to cover all the locations of $loc(\pi)$. However, ϕ is *not necessary*: different path conditions $\phi', \phi'', \phi''', \ldots$ not entailing ϕ can equally cover $loc(\pi)$. Think for example of Fig. 2: statement $\langle StmtB \rangle$ can be reached via $\pi = \langle \ell_1^1, \ell_1^0, \ell_2^0 \rangle$ (the while loop is executed once) with associated path condition $\phi = \widetilde{y} > 0 \wedge \widetilde{y} - 2 \leq 0$, and $\pi' = \langle \ell_1^1, \ell_1^1, \ell_1^0, \ell_2^0 \rangle$ (the while loop is executed twice) with associated path condition $\phi' = \widetilde{y} > 0 \wedge \widetilde{y} - 2 > 0 \wedge \widetilde{y} - 4 \leq 0$. Both these paths cover the same program points: $loc(\pi) = loc(\pi')$, even if $\pi \neq \pi'$ and $\phi \wedge \phi' \models \bot$.

The output of symbolic execution for a given program is a set Θ of symbolic traces, each of which corresponds to a path from the entry point of the program to its end point (including truncated paths if a termination criterion, e.g., a timeout, is met). We can define the set $\mathbb{T}(\Theta)$ of all the *concrete traces* corresponding to Θ as:

$$\mathbb{T}(\Theta) = \bigcup_{(\pi,\sigma,\phi)\in\Theta} \left\{ (x,\ell) \mapsto \text{ if } \ell \in loc(\pi) \text{ then } [\![\overline{\sigma}(x)]\!]_\xi \text{ else } \top \;\middle|\; \xi \models \phi \right\}$$

where $[\![\cdot]\!]_\xi : Exp \to Val$ is recursively defined by:

$$\begin{aligned} [\![c]\!]_\xi &= c && \text{for each } c \in Val \\ [\![x]\!]_\xi &= \xi(x) && \text{for each } x \in Var \\ [\![f(e_1,\ldots,e_k)]\!]_\xi &= f([\![e_1]\!]_\xi,\ldots,[\![e_k]\!]_\xi) && \text{for each } k \in \mathbb{N}, f \in Fun^k \end{aligned}$$

Each symbolic trace $(\pi, \sigma, \phi) \in \Theta$ can be unfolded into k concrete traces following π, where k is the (possibly infinite) number of solutions of path condition ϕ. Note that if a location ℓ is uncovered by a symbolic trace, we are not able to say that ℓ is unreachable in general because, as seen above, symbolic execution may miss concrete states. The set $\mathbb{T}(\Theta)$ is therefore an *under-approximation* of the set of all the feasible concrete states for a given program, i.e., there may exist a feasible concrete trace $\tau \notin \mathbb{T}(\Theta)$.

3.1 Concolic Testing

Historically, symbolic execution was suggested as a way of generating compact suites of test inputs, i.e., small sets producing large coverage. However, a number of issues have hindered its spread. Among these, we mention:

1. the source program may call library functions or make system calls
2. the underlying constraint solver may not be efficient or expressive enough to solve a given path condition
3. even simple programs tend to generate huge numbers of paths.

Concolic testing (or dynamic symbolic execution) was proposed [20] to overcome the first issue. Concolic is a portmanteau for concrete/symbolic, as concolic testing is a hybrid, maintaining both concrete and symbolic states, while executing a program. Thus, it has to be *seeded* with concrete values for symbolic variables. As it executes, concolic testing records alternative path constraints that can lead to new execution paths. A constraint solver is used to decide which ones are feasible and to provide new concrete inputs for the next path to explore.

Formally, we can define a *concolic trace* as a quadruple $(\rho, \pi, \sigma, \phi)$ where $\rho : Var \to Val$ is a *concrete state* and (π, σ, ϕ) is a *symbolic trace*. The concrete state ρ defines assignments of concrete values to symbolic variables. Each $\mathcal{L}$ assignment $x \leftarrow \top$ denoting a symbolic variable x is repeatedly replaced with a concrete assignment $x \leftarrow \rho(x)$, where the value $\rho(x)$ is decided at each iteration of the concolic testing according to the path condition found in the previous concolic iteration.

The semantics of concolic testing is very similar to that of symbolic execution. The main difference is that we actually execute the program, because symbolic variables now also take concrete values. So, the concolic execution is purely *deterministic*. The constraint solver is not used to evaluate the conditions at each branch point, because they are evaluated at runtime while executing the program. Instead, it is used to *generate* the concrete values to be assigned to the corresponding symbolic variables in the next concolic iteration.

Path conditions ϕ are recorded as for the symbolic execution. At the end of the concrete execution, a constraint C is removed from ϕ (typically the last). Then, a constraint solver is used to solve $\phi \wedge \neg C$. If there is a solution, we have new input values that will be used to feed the symbolic variables at the next iteration. Otherwise, the concolic process backtracks and a new constraint, not already negated, will be flipped. This process is repeated until all the constraints of ϕ are negated (or a termination criterion is met).

▶ **Example 2.** Let us see how concolic testing works on the $\mathcal{L}$ program in Fig. 2. Let us suppose that initially the value 0 is assigned to symbolic variable y (i.e., $\rho(y) = 0$). Both the conditions of the while loop and the "if" statement evaluate to false, so the concrete execution reaches statement $\langle StmtB \rangle$ with path condition $\phi = \neg(y > 0)$ (condition $x > 1117$ is not considered because x is not symbolic).

Then, $\neg(y > 0)$ is negated and a constraint solver will be used to solve $y > 0$. Assuming it returns the solution $y = 1$, we have a new concrete state (i.e., $\rho(y) = 1$) and hence we repeat the concrete execution by setting $y \leftarrow 1$. In this case, the loop is executed exactly once and $\langle StmtB \rangle$ is reached again, but this time with path condition $\phi = y > 0 \wedge \neg(y - 2 > 0)$. So we flip $\neg(y - 2 > 0)$ (we cannot flip $y > 0$, because it was already negated) and we solve $y > 0 \wedge y - 2 > 0$.

Assuming the solver returns the solution $y = 3$, we repeat the concrete execution by setting $y \leftarrow 3$ and so on, until all the constraints have been negated or the termination criterion is met. Note that the convergence of concolic testing also depends on the generated solutions. For example, if the solution of $y > 0$ was $y = 5000$ we would have reached $\langle StmtA \rangle$ right after the first iteration (by executing, however, the while loop 2500 times). ⌟

Concolic testing yields a set of concolic traces Γ containing both the concrete states and the symbolic traces for each explored path. From this point of view, concolic testing generalizes symbolic execution and we may define the concrete traces corresponding to Γ as $\mathbb{T}(\{(\pi, \sigma, \phi) \mid (\rho, \pi, \sigma, \phi) \in \Gamma\})$. However, often the symbolic computation is only used to generate the next input, so the path conditions can be overwritten at each concolic iteration – there is no need to keep track of them. We can therefore consider the output of concolic testing simply as $\{\rho \mid (\rho, \pi, \sigma, \phi) \in \Gamma\}$.

Concolic testing can be seen as an under-approximation of symbolic execution because it only considers a *witness* for each explored path, i.e., a solution for each path condition. Instead, symbolic execution defines *invariants* over program paths, i.e., necessary and sufficient conditions for the paths to be feasible. The ultimate goal of concolic testing is not to characterize the paths explored, but to maximize the overall *coverage* of a program, which we can define as the set $\bigcup_\pi loc(\pi)$ of all the locations traversed by any explored path π of Γ.

Concolic testing alleviates some of the issues of symbolic execution. Indeed, we can perform concolic testing without modelling library functions, system calls or third-party software by *directly invoking* all the functions. Constraint solving is more lightweight because constraints are simplified by the replacement of symbolic variables with concrete values. Importantly, we can simply *ignore* or approximate unsupported constraints. In this way, completeness is sacrificed but what is gained is that symbolic execution can proceed more smoothly. In many practical applications this is totally acceptable, provided that "good enough" coverage is achieved in reasonable time.

Thanks to a greater simplicity and efficiency w.r.t. symbolic execution, as well as the progress made by constraint solvers over the last years, concolic testing as become increasingly popular, and several concolic testing frameworks have been developed (e.g., DART [20], CUTE [34], jCUTE [33], KLEE [10]). The most recent tools for symbolic execution of C achieve considerable performance gains, for example by utilising tools that can perform *runtime instrumentation of binary code* (QSYM [38]), albeit at the cost of architecture dependence, or by *compiling* symbolic execution directly into LLVM bitcode (SymCC [29]), rather than interpreting bitcode, as done by KLEE.

4 Abstract Interpretation

Abstract interpretation is a well-established framework for static analysis. It is commonly used for *invariant generation*, i.e., for inferring program properties that hold for each possible program execution. To do so, it makes use of *abstract domains* for approximating sets of concrete runtime states.

Abstract interpretation is in a sense dual to symbolic execution. Indeed, plain symbolic execution under-approximates the set of possible concrete traces and can prove *reachability* (a semi-decidable problem in general). Analyses based on abstract interpretation usually over-approximate the set of concrete states, which means they can sometimes prove *unreachability*. However, one thing in common between these techniques is that they both, implicitly or explicitly, collect constraints along their execution. For symbolic (and concolic) execution, this aspect is evident. For abstract interpretation, we just have to make a little effort to see that abstract domains actually represent constraints defining invariant properties over the program variables.

For example, if the abstract value for an integer variable x is $Even$, the associated constraint is $x = 2k$ with $k \in \mathbb{Z}$. The direct product implicitly defines constraint *conjunctions*, e.g., $(Even, Pos, [-3, 8])$ corresponds to the constraint $x = 2k \wedge x > 0 \wedge -3 \leq x \leq 8$, whose feasible solutions are $\{2, 4, 6, 8\}$. A reduced product would refine these constraints into $x = 2k \wedge x > 0 \wedge 2 \leq x \leq 8$. In the constraint solving world, we are typically only interested in finding a feasible solution. Abstract interpretation aims instead to find a minimal set of constraints wrapping *all* the feasible solutions. The abstract domains define what *type* of constraints we are allowed to use to describe the invariants.

The link between constraints and abstract interpretation is clearer when we consider *relational* abstract domains. The simple non-relational domains seen above are efficient and easy to implement, but do not take into account the relations between variables, and thus tend to be imprecise. Relational domains instead are actually constraints capturing the relations between different variables. Examples of well known relational domains for numeric abstractions include linear congruence [22], octagons [27], and convex polyhedra [16]. For instance, the domain of convex polyhedra uses linear constraints of the form $a_1x_1 + \cdots + a_nx_n \leq b$, where the x_i are variables and the a_i, b are constants, to model the invariants. This domain offers considerable precision but also has high computational complexity (exponential in the worst case).

Formally defining a semantics as done in Section 3 capturing the abstract execution is not easy because there can be different ways of conducting the analysis and computing the invariants. For example, abstract domains can change dynamically (e.g., the CEGAR approach guides abstraction refinement via counter-example generation [11]). Moreover, unlike symbolic execution, abstract interpretation is not necessarily executed in a forward way. Instead of following the rules of an operational semantics, it effectively reasons *about* the execution flow of the program.

If *Abs* is a collection of abstract domains, we can define the *abstract state* for a given program as a map $Var \to \bigcup Abs$ from program variables to abstract elements. We denote the *abstract trace* with a pair (δ, λ) where $\delta : Var \to Abs$ maps program variable x to the chosen abstract *domain* $\delta(x)$ for it (including domain products and relational domains) and $\lambda : Var \times Loc \to \bigcup Abs$ returns the abstract *element* $\lambda(x, \ell) \in \delta(x)$ for x at program point ℓ (for simplicity, we assume that δ never varies during the program analysis). If variables $x_1, \ldots, x_k$ are abstracted with the same relational domain, then $\delta(x_1) = \cdots = \delta(x_k)$ and $\lambda(x_1) = \cdots = \lambda(x_k)$.

For example, given variable x and location ℓ, if $\delta(x)$ is the domain of intervals we may have $\lambda(x, \ell) = [-2, 7]$ but not $\lambda(x, \ell) = Even$. If $\delta(x) = Parity \otimes Sign$, a valid abstract element is $\lambda(x, \ell) = (Odd, NotPos)$. If both $\delta(x)$ and $\delta(y)$ are the octagon domain, we can have $\lambda(x, \ell) = \lambda(y, \ell) = \{x + y \leq 3, -x + y \leq 0, -x \leq 5, y \leq -1\}$ while, e.g., $\{x^2 + y^2 \leq 1\}$ is not a valid octagon.

We can define the *abstract execution* as the process of deriving *the* abstract trace (δ, λ) for a given program. In fact, here we can have at most one abstract trace per program, and not a collection of traces as happens for symbolic and concolic execution. This holds because the abstract trace over-approximates *all* the feasible concrete traces of the program. At a high level, abstract execution for $\mathcal{L}$ follows the control flow graph and updates (δ, λ) according to the statement encountered, e.g.:

- Initially $\lambda(x, \ell) = \bot_{\delta(x)}$ for each program variable x and location ℓ, except for the location ℓ_0 of the entry point of the program, for which $\lambda(x, \ell_0) = \top_{\delta(x)}$.
- For (ℓ) $x \leftarrow e$ (ℓ'), the abstract value $\lambda(x, \ell')$ is defined according to the transfer function for the concrete expression e. For relational domains, we may also update $\lambda(y, \ell')$ for each variable y occurring in e.
- For (ℓ) **if** b **then** (ℓ_1) S_1 (ℓ_1') **else** (ℓ_2) S_2 (ℓ_2') **fi** (ℓ_3), condition b is used to refine the abstract elements $\lambda(x, \ell_1)$ and condition $\neg b$ is used to refine the abstract elements $\lambda(x, \ell_2)$ for each variable x involved in b. Once S_1 and S_2 are processed, the control flow merges and so we join the abstract value of each variable occurring in S_1 or S_2: $\lambda(x, \ell_3) = \lambda(x, \ell_1') \sqcup_{\delta(x)} \lambda(x, \ell_2')$.
- For (ℓ) **while** b **do** (ℓ_1) S (ℓ_1') **od** (ℓ_2), condition b is used to refine the abstract elements $\lambda(x, \ell_1)$ and then the widening operation $\lambda(x, \ell_1) \nabla_{\delta(x)} \lambda(x, \ell_1')$ is repeatedly applied until a fixpoint is reached. Condition $\neg b$ is used to refine $\lambda(x, \ell)$, and then the join operation is applied between such refined abstract element and the "stationary' element computed by ∇.

▶ **Example 3.** Consider again the $\mathcal{L}$ program in Fig. 2. Suppose that for variables x and y we chose the sign domain and the interval domain respectively. At location ℓ_1, just before the while loop, we have $\lambda(x) = Zero$ and $\lambda(y) = \top$. At location ℓ_2, just before the if-then-else statement, the abstract execution is able to determine that we have $\lambda(x) = NotNeg$ (i.e., $x \geq 0$) and $\lambda(y) = (-\infty, 0]$. However, a more precise analysis would be able to infer that $y \in [-1, 0]$. Even more precisely, a relational analysis may detect the invariant $2x + y - y_0 = 0$ where y_0 is the value of y before entering the while loop. ⌟

Each invariant $\lambda(x, \ell)$ is to all effects a constraint over the possible values that x can take at location ℓ. The type of this constraint is defined by $\delta(x)$: we have unary constraints for non-relational domains, conjunctions of constraints for domain products, and (conjunctions of) k-ary constraints for relational domains involving k variables. If we denote with $[\![a]\!]_{\mathcal{A}}$ the constraint corresponding to an abstract element $a \in \mathcal{A}$, we can define the set of the concrete traces corresponding to an abstract trace (δ, λ) as:

$$\mathbb{T}(\delta, \lambda) = \{(x, \ell) \mapsto \xi(x) \mid \xi \models [\![\lambda(x, \ell)]\!]_{\delta(x)}\}.$$

In practice, for each $x \in \mathit{Var}$ and $\ell \in \mathit{Loc}$, we associate to the pair (x, ℓ) the concrete value $\xi(x)$, where ξ is a solution for the constraint corresponding to the abstract element $\lambda(x, \ell)$.

As mentioned, abstract execution returns a single abstract trace for each program. In general, if $\mathbb{T}^*$ is the set of all the feasible concrete traces for a given program, Θ is the set of the symbolic traces resulting from a symbolic execution of that program, and (δ, λ) is the abstract trace resulting from its abstract execution, we have that:

$$\mathbb{T}(\Theta) \subseteq \mathbb{T}^* \subseteq \mathbb{T}(\delta, \lambda).$$

In other words, symbolic execution under-approximates the feasible concrete states, while abstract interpretation over-approximates them. Again, the assumption made here is that symbolic execution operates without assistance from oracles or added induction tools. Moreover, we assume that the underlying abstract execution is *sound*.

5 Synergy

In this section we bring things together by discussing some synergies between the worlds of symbolic execution, abstract interpretation, and constraint solving. In particular, we show how abstract interpretation can enrich symbolic execution through program transformation.

5.1 Abstract interpretation and constraint solving

As seen in Section 4, there is an implicit bond between abstract interpretation and constraint solving. Because the invariants computed by abstract interpretation are actually constraints over-approximating concrete states, constraint solvers can be used to improve the precision of the analysis by *refining* the abstract domains. Constraint solving can soundly rule out infeasible configurations and possibly detect unsatisfiability. Moreover, it can be used to generate counterexamples. For example, Ponsini, Michel and Rueher [30] present a hybrid approach for the abstract interpretation of floating-point programs using constraint programming to tighten the abstract domains and therefore reduce the number of false alarms.

What constraint solving can learn from abstract interpretation is the use of abstract domains to represent the domain of the decision variables and the relations between them – especially for non-trivial, structured types. For example, the dashed string abstraction introduced in [3] for string constraint solving is based on the Bricks abstract domain introduced in [13] for the analysis of strings.

5.2 Symbolic execution and constraint solving

Constraint solving and symbolic (and concolic) execution are strongly coupled because path conditions are iteratively solved with an underlying constraint solver. The expressiveness and the efficiency of the solver have huge impact on the performance of the symbolic execution: the

better the solver, the better the symbolic execution. Arguably, the remarkable improvements of constraint solvers over the last decades positively affected the development of symbolic and concolic execution frameworks.

Symbolic execution can also help the development of new and better constraint solvers. The path conditions arising from program analysis can suggest the development of new types of variables, constraints and search heuristics (e.g., aggregate types and complex operations) and be used as benchmarks – generated for free from the program source – to validate and evaluate constraint solvers.

5.3 Abstract interpretation and symbolic execution

Somehow, abstract interpretation is performed by "symbolically" executing the input program, not necessarily in a forward way, using abstract values instead of the concrete ones. In principle, because it works by over-approximations, it can only generate false positives. In practice, unsound domains are sometimes used: in this way, we can also have false negatives. Symbolic (or concolic) execution might be used to *post-process* the abstract execution via counterexample generation w.r.t. a property ϕ of interest, provided that we can encode that ϕ with a corresponding Boolean expression b in the input language. In this way, we can check if ϕ (or $\neg\phi$) holds at program point ℓ by inserting at that point an "if-then-else" construct having guard condition b.

What is probably more interesting is instead the other way round: how abstract interpretation can help symbolic execution.

For example, a common problem for "classical" symbolic execution, as we defined it in the previous sections, is that it often gets stuck exploring loops for an intolerably long time. This happens because usually symbolic execution makes no attempt to reason about a program – all it does is following the rules defined by the transition system of its operational semantics. Abstract interpretation, on the other hand, steps outside of the operational semantics of symbolic execution by reasoning *about* loops rather than simply executing them.

Abstract interpretation can be used *transform* the source program in a target program where symbolic execution can escape loops. For example, *loop counters* can sometimes aid the abstraction enhanced symbolic execution [2]. The addition of counters does not interfere with the semantics of the program – it is essentially a semantics-preserving transformation. The kind of source-to-source transformations that aim to preserve some testing metric (but possibly not the semantics of the program) are called *testability transformations* [23]. Examples include the merging or splitting of loops, induction variable substitution, changing the type of a variable from float to int, and others [23].

A possible way of enriching the symbolic execution of the $\mathcal{L}$ language is to augment it with *assumptions*, whose purpose is to actually *replace* the while loops with the invariants that abstract interpretation yields at the end of each loop. In practice, for transforming while loops into corresponding "assume" statements, we replace rule 6 of Fig. 1 with:

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad b' = \bigwedge_{x \in S} [\![\lambda(x,\ell)]\!]_{\delta(x)} \quad \sigma'(x) = \begin{cases} \widetilde{x} & \text{if } x \in S \\ \sigma(x) & \text{if } x \notin S \end{cases}}{\langle \textbf{while } b \textbf{ do } S \textbf{ od } (\ell), (\pi, \sigma, \phi)\rangle \to \langle (\ell) \textbf{ assume } b', (\pi, \sigma', \phi)\rangle}$$

where $b' = \bigwedge_{x \in S} [\![\lambda(x,\ell)]\!]_{\delta(x)}$ is the conjunction of all the constraints corresponding to the invariant $\lambda(x,\ell)$, for each variable x occurring in statement S. Note that this rule does not modify either the path π or the path condition ϕ. However, map σ is updated by assigning a new, fresh variable $\widetilde{x}$ to each variable x occurring in statement S. This somehow has the effect of forgetting the history of these variables, in order to avoid conflicts with the introduced invariant b'.

The rule for the assume statement is simply defined as:

$$\frac{\phi \wedge \overline{\sigma}(b) \not\models \bot \quad \phi' = \phi \wedge \overline{\sigma}(b) \quad \pi' = \pi \oplus \ell^1}{\langle (\ell)\ \textbf{assume}\ b, (\pi, \sigma, \phi)\rangle \rightarrow (\pi', \sigma, \phi')}$$

Clearly this kind of loop elimination can introduce false positives, but it allows symbolic execution to escape loops (the more precise the abstract execution is, the less likely this is to generate false positives). This transformation relies on the fact that test data generation is a *forgiving* application: the possibility of *path divergence* [4] is already a reality in concolic testing, and the damage risked is simply the generation of sub-optimal test sets. One can also consider a parametric approach where the while loop is executed at most k times: if after k iterations the loop condition still holds, then the while loop is transformed.

▶ **Example 4.** Consider once again the $\mathcal{L}$ program in Fig. 2. Let us suppose that the abstract interpretation is conducted with the interval domain, so after the while loop we have the following invariants: $\lambda(x, \ell) = [0, +\infty)$ and $\lambda(y, \ell) = (-\infty, 0]$ where ℓ is the location corresponding to the end of the loop (line 6). If we apply the loop elimination described above, we set $\sigma = \{x \leftarrow \widetilde{x}, y \leftarrow \widetilde{y}\}$ and we replace the while statement with: (ℓ) **assume** $x \geq 0 \wedge y \leq 0$.

Then, the assume statement is evaluated and the symbolic execution can proceed with the evaluation of the "if-then-else" statement. Clearly, this approach ensures the termination of the symbolic execution but it does not guarantee its soundness. For example, the evaluation of the "if" condition $x > 1117$ can succeed with solution $\widetilde{x} = 1118$, $\widetilde{y} = 0$ not corresponding to any feasible concrete state. However, with more precise (and relational) domains many of these spurious configurations can be ruled out (e.g., by adding the invariant $2\widetilde{x} + \widetilde{y} - y_0 = 0$ where y_0 is the value of y before executing the while loop) ⌟

6 Related Work

A number of systems for program verification have been based on symbolic execution. VeriFast [24] uses separation logic to verify various properties of (subsets of) C and Java, including properties of functions that manipulate inhabitants of recursively defined data types such as lists and trees.

The KeY project [6] (`https://www.key-project.org/`) is an active Java program verification project. KeY uses a sequent-based dynamic logic. Users can provide method contracts and loop invariants, with system support for checking the validity of invariants. The system can then use the invariants in place of the loops that are thus abstracted. This may simplify the verification task, and solve the problem of symbolic execution getting caught in loops (naturally, if the abstraction is too imprecise, it may also fail to enable a desired proof). Abstraction can also be used to model components for which the source code is unavailable. For consistency, such models must over-approximate the possible runtime states (in contrast to the "concretizing" approach used by concolic testing tools).

Of particular interest in our context is the integration of abstract interpretation with KeY [37]. The symbolic execution based reasoning about the values that a variable can take is interleaved with reasoning about value sets (which are elements of abstract domains), and as a result, some invariants can be generated automatically. The abstract domains may be refined in the process. From available descriptions, it appears that *values*, rather than program states, are abstracted (so that relational analysis is excluded).

It is interesting to compare the KeY project's use of abstraction in symbolic execution based verification with the use we suggest for concolic testing. A static analysis that provides "attribute independent" (or non-relational) abstraction can be of great value in the verification

context, but it is of little use to concolic testing. To be of any value, an oracle for a concolic testing tool has to be able to provide non-trivial information about how the values of different variables are related. Fortunately, the availability of sophisticated tools for abstract interpretation allows a clean separation of concern, effectively providing us with a highly parametric oracle.

Another application with a very focused aim is the static analysis used by Feist et al. [18] to identify "use after free" vulnerabilities in a binary-code. Static analysis is used to compute "weighted slices" which are then used to guide DSE. The weight attached to a slice reflects the degree to which the slice is able to include an "allocate then free then use" pattern for some pointer variable.

Shastry et al. [35] apply static analysis to improve non-grammar based fuzzing. A simple static analysis is used to construct better fuzzing input dictionaries. Normally such dictionaries are created based on string literals found in the program under analysis. Shastry et al. show how a kind of taint analysis combined with backward slicing can help generate more effective dictionaries in the context of network applications.

Adding loop counters as a benign transformation to facilitate better analysis is also seen in work on abstract domains [36] and in symbolic execution. For example, the "loop-extended" grammar-based symbolic execution proposed by Saxena et al. [32] involves adding a new symbolic variable (or *trip count*) per loop. The trip count is related to the program's input format through further auxiliary variables, to capture how variables assigned in loops depend on the lengths and counts of elements in the program input. Godefroid and Luchaup [21] identify unbounded loops that use induction variables (in a linear manner). These loops are summarized by pre- and post-conditions, derived from inferred partial loop invariants, relating program inputs to the values of induction variables. No static analysis is involved in this.

7 Discussion

This paper has explored interactions between abstract interpretation, symbolic execution and constraint solving. We exposed the – sometimes implicit – bond that constraints have with both symbolic execution and abstract interpretation, and we put forward a view that symbolic execution is best considered a dynamic analysis.

We discussed a way of helping concolic testing escape loops via abstract interpretation and program transformation. In a context of under-approximating dynamic analysis it may, at first, seem surprising that an over-approximating static analysis can be of help. Indeed, it can be of help only because it can approximate sets of program states, rather than simply sets of values. That is, unlike the dynamic analysis, the static analysis can contribute *relational* information. We believe that this tells us something new and important about the nature of dynamic vs static analysis.

A variety of characterizations and definitions can be found in the literature, the most common one being that static analysis is an analysis that is valid for any execution of the program. In a similar vein, Tom Ball [5] characterises static analysis as "*program centric*" as opposed "*input centric*" dynamic analysis. He also sees the higher precision of information gained through dynamic analysis as an important characteristic.

We suggest that another important distinction is between what may be termed *value oriented* and *state oriented* analysis. Dynamic analysis, symbolic execution included, is usually value oriented: a "symbolic" runtime state is a mapping from variables to symbolic expressions, the latter describing sets of concrete *values*. Abstract interpretation, in contrast,

bases itself on a "collecting semantics", associating possible runtime states with each program point. An "abstract" runtime state is sometimes designed as a mapping from variables to abstract values (describing sets of concrete values), but it does not have to be a mapping. It could be a more fine-grained description in the form of a *relation* that describes a set of concrete *runtime states*. We have exemplified how this can be used to improve applications of symbolic execution, such as concolic testing. The only relational information considered by a concolic testing tool is how program variables may depend on input, that is, how they are related to symbolic variables (as expressed through path constraints). A static analysis can expose more complex relations between variables, thus providing additional information about runtime states – information that, for example, a test-data generating constraint solver can utilise.

In future work we would like to revisit the points made here, and develop a proper framework that can serve as a formal foundation for this discussion.

References

1 W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Dedecutive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

2 Eman Alatawi, Harald Søndergaard, and Tim Miller. Leveraging abstract interpretation for efficient dynamic symbolic execution. In G. Rosu, M. Di Penta, and T. N. Nguyen, editors, *Proc. 32nd IEEE/ACM Int. Conf. Automated Software Engineering*, pages 619–624. IEEE Comp. Soc., 2017.

3 Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. A novel approach to string constraint solving. In J. C. Beck, editor, *Proc. 23rd Int. Conf. Principles and Practice of Constraint Programming*, volume 10416 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017. doi:10.1007/978-3-319-66158-2_1.

4 Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.

5 Thomas Ball. The concept of dynamic analysis. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering – ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1999.

6 Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic logic for Java. In *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*, pages 49–106. Springer, 2016.

7 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

8 Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT: A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10(6):234–245, 1975.

9 R. M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing: Proc. IFIP Congress 1974*, pages 308–314. North-Holland, 1974.

10 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, volume 8, pages 209–224, 2008.

11 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification: Proc. 12th Int. Conf.*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

12 Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures*, 37(1):24–42, 2011.

13 Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. A suite of abstract domains for static analysis of string values. *Software Practice and Experience*, 45(2):245–287, 2015.

14 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

15 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages (POPL'79)*, pages 269–282. ACM, 1979.

16 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear constraints among variables of a program. In *Proc. Fifth ACM Symp. Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.

17 George B. Dantzig. Linear programming. *Operations Research*, 50(1):42–47, 2002.

18 Josselin Feist, Laurent Mounier, Marie-Laure Potet, Sébastien Bardin, and Robin David. Finding the needle in the heap: Combining static analysis and dynamic symbolic execution to trigger use-after-free. In *Proceedings of the 6th ACM Workshop on Software Security, Protection, and Reverse Engineering*, pages 2:1–2:12. ACM, 2016.

19 Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Abstract interpretation over non-lattice abstract domains. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2013.

20 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005. doi:10.1145/1065010.1065036.

21 Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. 2011 Int. Symp. Software Testing and Analysis (ISSTA'11)*, pages 23–33. ACM, 2011. doi:10.1145/2001420.2001424.

22 Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91: Proc. Int. Joint Conf. Theory and Practice of Software Development, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

23 Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

24 Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the VeriFast program verifier. In K. Ueda, editor, *Programming Languages and Systems: Proc. 8th Asian Symp.*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010.

25 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

26 Jean-Louis Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intellgence*, 10(1):29–127, 1978.

27 Antoine Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

28 Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

29 Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *Proc. 2020 USENIX Security Symp.* USENIX, 2020. Could not find this on the USENIX Security 20 web site.

30 Olivier Ponsini, Claude Michel, and Michel Rueher. Verifying floating-point programs with constraint programming and abstract interpretation techniques. *Automated Software Engineering*, 23(2):191–217, 2016.

31 F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

32 Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA'09)*, pages 225–236. ACM, 2009. doi:10.1145/1572272.1572299.

33 Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification: Proc. 18th Int. Conf.*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

34 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proc. 10th European Software Engineering Conf.*, pages 263–272. ACM, 2005. doi:10.1145/1081706.1081750.

35 Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses: Proc. 20th Int. Symp. (RAID'17)*, volume 10453 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2017.

36 Arnaud J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In P. Madushan and S. A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.

37 Nathan Wasser, Reiner Hähnle, and Richard Bubel. Abstract interpretation. In *Deductive Software Verification – The KeY Book*, volume 10001 of *Lecture Notes in Computer Science*, pages 167–189. Springer, 2016.

38 Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proc. 27th USENIX Security Symp.*, pages 745–761. USENIX, 2018.